**הרצאות 12 – קורס קודם**

**Serialization:**

* הרעיון של לקחת אובייקט ולהעביר אותו ממקום למקום – יש לנו אובייקט עם מצב פנימי ואנו רוצים לקחת אותו ולשמור אותו במקום מסוים, לעשות גיבוי, להעביר אותו למישהו וכדומה.
* הFRAMEWORK שנעבור בו הוא STREAMS (צינורות דרכם המידע עובר) – נסתכל על אובייקט כמידע ונעביר אותו כSTREAM
* נניח שיש לנו לקוח עם שדות ומידע עליו ואנו רוצים לקחת אובייקט של לקוח ולשמור אותו (אותו ואת כל אחד מהערכים שלו) –
  + נתחיל בלעבור על השדות שלו ולשמור אותם
  + כאשר מדובר על רשימות ודברים מורכבים יותר אנחנו צריכים להעתיק ממש את האובייקטים האלו – רקורסיבי
  + מה קורה אם אנחנו שומרים משהו והוא משתנה תוך כדי?
* סריליזציה זה התהליך של לקחת אובייקט (של ג'אווה) ולהעביר אותו לאיזשהו צינור מידע STREAM – תהליך זה עובד בצורה רקורסיבית, כלומר על כל שדה של האובייקט נריץ שוב תהליך זה
* דה-סריליזציה הוא התהליך ההפוך – לקחת אובייקט שנכתב לסטרים לקרוא אותו ולשחזר ממנו את האובייקט המקורי
* STREAM יכול להיות למשל דיסק
* מונחים שוקלים – serializing out, serializing in
* שמירת אובייקטים -
  + האובייקט צריך להסכים שישמרו אותו בSTREAM – אנו שומרים למתכנתים את האפשרות לא להסכים שישמרו את המחלקה שלהם
  + כל מחלקה בג'אווה דיפולטיבית לא מסכימה שישמרו אובייקטים שלה
  + כדי שמחלקה תסכים היא צריכה לממש ממשק Serializable שהוא ריק (אין בו מתודות) – כל מה שהוא עושה זה לסמן שהמחלקה מסכימה לשמור אובייקטים שלה
    - ממשקים ריקים נקראים marker interface – ממשק שכל תפקידו הוא לסמן
  + בגלל שהתהליך רקרוסיבי כל אחד מהשדות של האובייקט, והשדות של השדות שלו (וכן הלאה) צריכים לממש את הממשק הזה או להיות פרימיטיביים (אפשר לשמור פרימיטיביים סתם ככה)
* איך התהליך נראה בג'אווה
  + תהליך הכתיבה (שמירת אובייקט) נעשה בעזרת מחלקה ObjectOutputStream ותהליך הקריאה נעשה בעזרת המחלקה ObjectInputStream
  + אלו מחלקות STREAM והן ספציפית מחלקות שעושות decorating – מרחיבות את STREAM ומוסיפות
  + ניצור אובייקט של OUTPUTSTREAM עם FILEOUTPUTSTREAM ואז ניצור אובייקט של OBJECTOUTPUTSTREAM וניתן לו את האובייקט הראשון. נפעיל עליו את המתודה writeObject שניתן לה את האובייקט לשמירה
  + בדומה כדי לקרוא ניצור INPUTSTREAM עם FILEINPUTSTREAM ואז ניצור OBJECTINPUTSTREAM וניתן לו את האובייקט הראשון. נפעיל את הפונקציה readObject ונעשה DOWNCASTING לטיפוס האובייקט אותו אנו קוראים לשניהם ניתן את אותה כתובת ביצירת הSTREAM

**Intricacies of Serialization:**

* תהליך הסריליזיציה הוא התהליך שלוקח אובייקט ושומר את כל גרף השדות שלו.
* מה קורה אם בתהליך אנו מגיעים לשדה שכבר שמרנו בעבר? אנחנו לא נשמור אותו שוב אלא נשמור מצביע למה ששמרנו כבר.
  + חשוב במצבים בהם יש לנו מעגל של אובייקטים שכל אחד מכיל שדה של האובייקט הבא במעגל – כך לא נתקע בלולאה אינסופית.
* האובייקטים נשמרים בFIFO – הראשון שנשמר הוא הראשון שיקרא
* אם שמרנו אובייקט, שינינו את אחד השדות, אז אם נרצה לשמור אותו שוב אנו נשמור למעשה מצביע לאובייקט הראשון ששמרנו (כלומר השינוי לא יישמר).
  + נשמור את השינוי רק אם קראנו כבר את האובייקט הראשון ששמרנו
  + כדי לעקוף את התהליך נוכל גם לסגור את הSTREAM בין השמירות (עם close()) או לקרוא ל-reset() שמוחקת את כל מה ששמרנו לפני
* שדות transient –
  + Transient היא מילה שמורה בג'אווה שניתן לשים לפני שדות מחלקה והיא אומרת שלא נשמור שדות אלו כאשר נשמור את האובייקט.
  + כך למעשה אנו יכולים להחליט לשמור רק שדות שרלוונטיים לשמירת האובייקט
  + כמו כן שדות סטטיים לא צריכים להיכלל בשמירת האובייקט כי הם לא ספציפיים לאובייקט
* טיפוסים פרימיטיביים –
  + אי אפשר לבצע תהליך סריליזציה על פרימיטיביים
  + נוכל לעשות זאת באמצעות DataOutput וDataInput שמאפשרים לשמור פרימיטיביים עם writeInt() לדוגמא
  + לא מדובר על שדות מחלקה פרימיטיביים אלא על אובייקטים פרימיטיביים שנרצה לשמור בתוך STREAM
* שינוי מחלקה –
  + נניח ששמרנו אובייקט לSTREAM ולאחר מכן ביצענו שינויים במחלקה.
  + כאשר נרצה לטעון אותו מחדש ניתקל בבעיה – יכול להיות שהיו שינויים שיהפכו את תהליך הdeseriliazation לבלתי אפשרי
  + אולם יש שינויים שכן יאפשרו לטעון מחדש את האובייקט
  + הפתרון – נשמור משתנה סטטי SerialVersionUID במחלקה – בתור כותבי המחלקה אנו יודעים אילו שינויים הם קריטיים ולא מאפשרים לטעון אובייקטים לפני השינוי.
    - כאשר נשמור אובייקט נשמור איתו את השדה של הגרסה.
    - כאשר נשנה את המחלקה עם שינויים שלא מאפשרים לטעון אובייקטים ישנים נשנה את משתנה הגרסה.
    - כאשר נרצה לטעון אובייקט נבדוק אם הגרסה השתנתה, כלומר אם ניתן לטעון את האובייקט למרות השינויים.
  + שדה זה הוא לא חובה – ג'אווה תמיד מגדירה שדה זה, אך ג'אווה חושבת שכל שינוי הוא בעייתי, כלומר על כל שינוי תשנה את הגרסה ולא תאפשר טעינה של אובייקטים ישנים גם אם השינויים לא משפיעים
  + תמיד כדי להגדיר את שדה הגרסה

**Cloning:**

* נרצה ליצור אובייקט חדש עם בדיוק אותם פרטים כמו של אובייקט קיים.
* העתקה שטוחה – יוצרים אובייקט חדש שכל השדות שלו מצביעים לשדות האובייקט המקורי
  + כלומר שינויים באובייקט החדש יגררו שינויים באובייקט המקורי ולהיפך
  + יש להם את אותם שדות מחלקה
* העתקה עמוקה – שדות שאינם פרימיטיביים מועתקים באופן עמוק רקורסיבית גם כן
  + כלומר אין לשני האובייקטים אותם שדות ושינוי של אחד לא יגרור שינוי של השני
* שכפול בג'אווה:
  + דומה לעניין הסריליזציה
  + אובייקט צריך להסכים שישכפלו אותו ע"י מימוש הממשק Clonable() שהוא ממשק מסמן marker interface
  + כדי שיוכלו לשכפל אותי יש דרישה לדרוס את המתודה clone() (של Object)
  + Clone() – כברירת מחדל יוצר שכפול שטוח, לכן חשוב לדרוס אותו בשביל שכפול עמוק
* Object.clone() – קודם כל בודקת שהאובייקט מממש את CLONABLE (אם הוא לא ממש את הממשק היא זורקת exception של CloneNotSupportedException). לאחר מכן מבצעת שכפול שטוח
* CLONE מחזירה Object ולכן צריך לעשות casting
* מערכים גם מממשים את Clonable אבל גם מבצעים שכפול שטוח
* הממשק clonable הוא לא מומלץ בשביל שכפול – עדיף להשתמש בcopy constructor
  + פשוט יותר
  + מאפשר לשכפל אובייקט מטיפוס שונה
* למעשה אנו יוצרים אובייקט חדש ומשכפלים את הערכים שלו מערכי האובייקט המקורי.
  + כלומר נצטרך לשכפל בעצמנו את השדות הרלוונטיים בשביל שכפול עמוק

**Reflections:**

* מנגנון שמאפשר לתוכנית או מחלקה להסתכל על עצמה, לבצע אינטרוספקציה – לענות על שאלות כמו מה הי, לשנות חלקים בתוכה
  + לדוגמא לקחת מחלקה ולהגיד לה להדפיס את כל המחלקות שלה.
* אחד הכלים הכי חזקים שנלמד, אבל גם אחד הכלים הכי מסוכנים
* מחלקת Class של ג'אווה –
  + כל אובייקט בג'אווה יכול להיות פרימיטיבי (מספר, בוליאני, תו...) או רפרנס (אובייקט, מחלקה, מערך, ממשק...)
  + לכל אובייקט שהוא רפרנס מאותחל אובייקט immutable של מחלקה שנקראת java.lang.class
  + גישה לאובייקט זה :
    - בעזרת המתודה הסטטית Class.forName(“class name”) עם שם המחלקה
    - ללכת לאובייקט מסוים שיש לו מחלקה מסוימת ולקרוא ל- myObj.getClass() ולשים בטיפוס מסוג Class
    - אם המחרוזת לא מייצגת מחלקה קיימת או שזו מחלקה שאין לנו גישה אליה ייזרק ClassNotFoundException
  + מה אפשר לעשות עם אובייקט Class שנקבל?
    - לקבל את רשימת הקונסטרקטורים שלו עם המתודה getDeclaredConstructors()
      * נקבל רשימה של Constructor
      * נוכל לעבור על הרשימה וליצור מופע מחלקה לפי הקונסטרקטורים עם <constructor[i]>.newInstance(<argslist>)
        + כאשר argslist נוצרת לפי הפרמטרים שמקבל הקונסטרקטור – getParameterTypes()
    - אפשר לקבל רשימה של כל המתודות של המחלקה – getDeclaredMethods() שמחזירה רשימה של Method
      * נקבל גם את המתודות הפרטיות של המחלקה
      * על כל מתודה אפשר לשאול איזו מחלקה הגדירה אתכן getDeclaringClass(), אילו פרמטרים מקבלת, מה היא מחזירה.
      * אפשר להריץ את המתודות בעזרת invoke על המתודה עם האובייקט (מופע המחלקה) ועם רשימת ארגומנטים – יוחזר ערך ההחזרה של המתודה עם up-cast ל-Object או null אם הוחזר void
    - אפשר לקבל שדות – getDeclaredFields() שמחזיר רשימה של Field
      * אפשר לעדכן אותם עם set (מקבלת את האובייקט ואת ערך השדה החדש שנרצה) או לקבל את מה ששמור בשדות עם get (מקבל את האובייקט)
      * אפשר לשאול מה הסוג שלו, מה הmodifiers
      * לא עובד דיפולטיבית על שדות פרטיים אך אפשר לגשת אליהם אם נרצה
* למה להשתמש בזה?
  + כלי שימושי להגדיל את הגמישות והרחבה של המחלקה
    - מאפשר הוספת מחלקות חדשות שהמחלקה המקורית לא הכירה בזמן קמפול
    - עוקף שימוש בבלוקי SWITCH ארוכים
  + כלי דיבוג
  + סריאליזציה – למחלקות הSTREAM יש גישה למידע פרטי של אובייקט של מחלקה אחרת בעזרת הREFLECTIONS
* חסרונות:
  + הולך נגד רעיונות האנקפסולציה והסתרת מידע
  + יכול ליצור תופעות לוואי שיהרסו את פונקציונליות הקוד או את האפשרות להעביר את הקוד
  + תוכניות שמשתמשות בזה הן משמעותית יותר איטיות
* תזכורת – טעות נפוצה של בלבול בין מידע פרטי למידע סודי –
  + מידע סודי לא צריך להישמר בשדה פרטי, אם נרצה להגן עליו צריך להשתמש בהצפנה
  + שימוש בפרטי – הרעיון הוא עיצוב יותר טוב – להחביא כמה שיותר פרטים כדי שלי יהיה חופש תוך שמירה על הAPI. REFLECTIONS מאפשרים לעקוף את PRIVATE וזה מסוכן.